My Project

Generated by Doxygen 1.9.1

Chapter 1

Introduction and Performance Analysis

1.0.1 How2Run

In order to successfully run, the executable of main.cpp and books.csv must be in the same directory. Commands that can be run are:

- g++ -g main.cpp
- ./a.out

sorted_books.csv will be created if the input file is present and thus the sorting is correct. If the sorting is wrong the output file will not be created.

1.0.2 Performance Analysis and Part2.c

The performance analysis includes gprof for profiling and gprof2dot for the visualisations. Gprof counts how many times each function is run by putting print statements around the functions. In order to use gprof, -pg flag must be given during compile time. When the executable is run a gmon.out file is created which gprof reads and creates a flat profile and a call graph in text. Gprof2dot takes text input and creates images of call graphs with execution information.

In the first run of my analysis, I realised that the parser was taking 50% of the execution time. The reason for that as it can be seen at Figure 1.1, is the vector.push_back function utilised the malloc() for vector significantly. As the input gets larger it created a growing overhead. To solve it I reserved the maximum expected size at the beginning and in the end of parsing made it shrink_to_fit() to not allocate more than necessary memory.

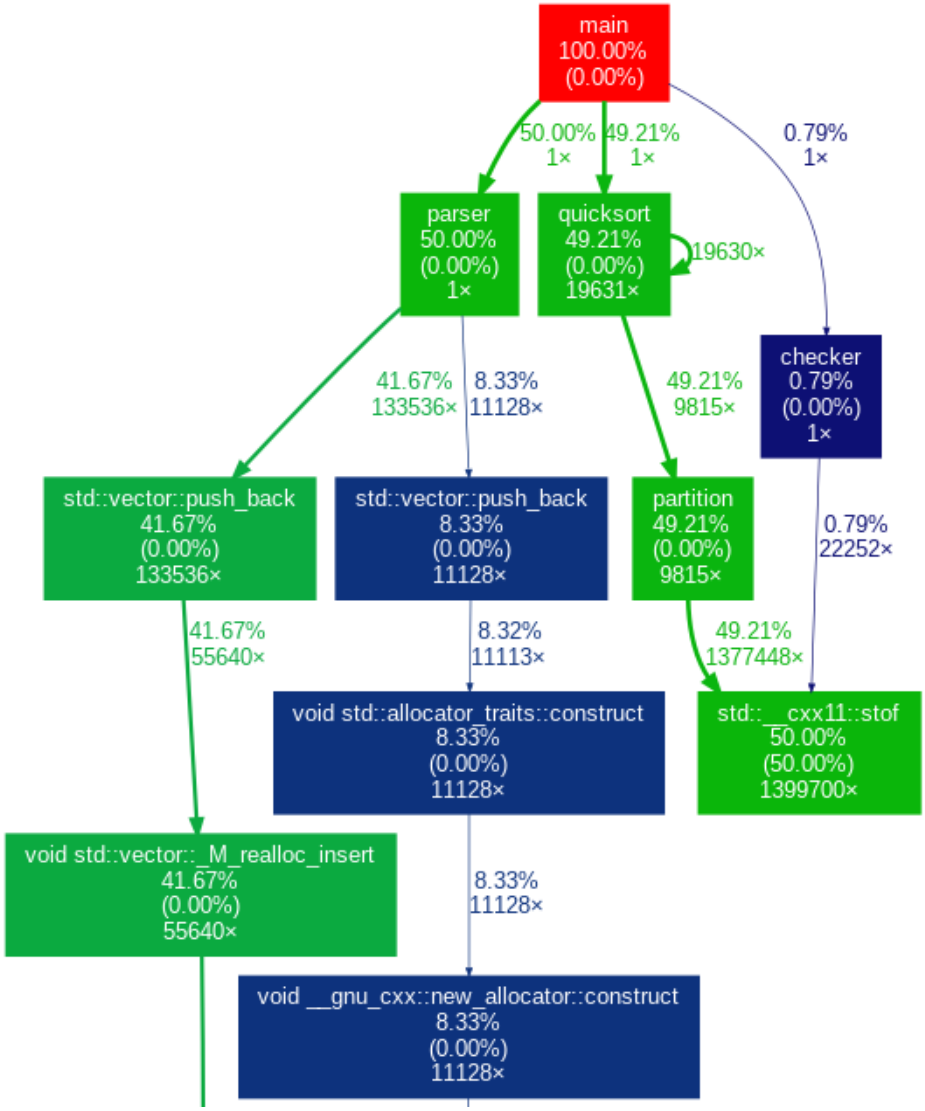


Figure 1.1 O2 optimised and compiled by g++

In my first iteration of the code, to see that the algorithm is working, the parsed vector was directly accessed and sorted by using string to float (stoa) function extensively. This caused an overhead as the complexity of the sorting algorithm got reflected in stoa() function as well which can be observed at Figure 1.2.

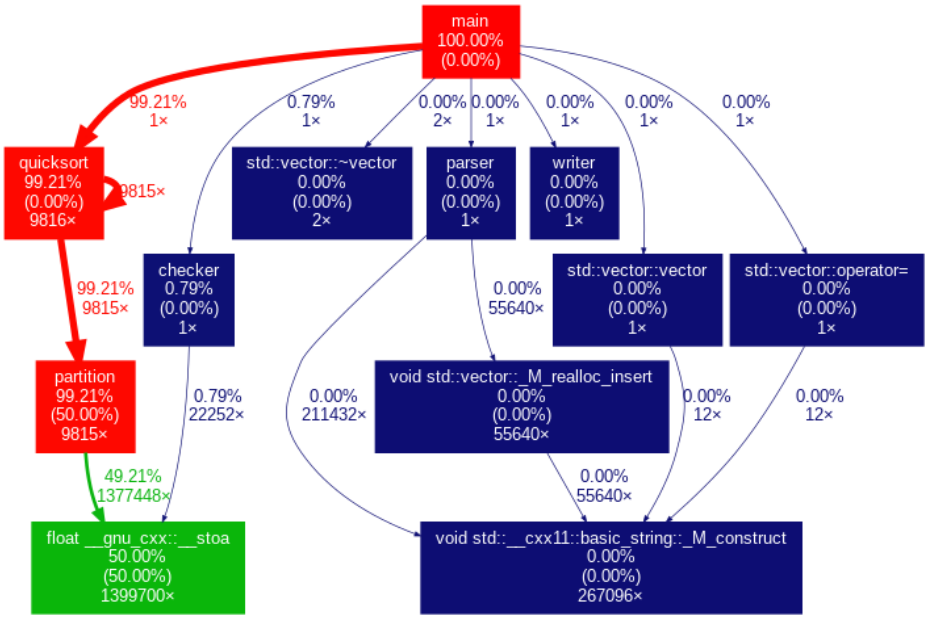


Figure 1.2 O2 optimised and compiled by g++

To remove the stoa() overhead, the reading of average rating number and the sorting of vectors must have been decoupled from the reading of the parsed vector. To achieve the decoupling two arrays of integers were considered. The unsorted_values

array included the essence of the read file. At every index i similar to row is the integer rating number. The read operation and comparison is performed on this array with the index provided from the proceeding array which holds the indices. The stof() operation is done once per row in this way in the extractor() function. To accomodate swap operations another array, sorted_keys

, holding the indices of the unsorted_values array is created. Initially Arr

$$i$$

= i as they are unsorted. To perform swap, two index values residing in sorted_keys array is swapped. To ensure the new index based sorting system is safe, a new class with these arrays is declared and made a global object. The class has a getValue() and swap operation. Sorting functions do not access the arrays directly.

After making these changes the parser() function is still taking 100% execution time as it can be seen at Figure 1.3. After investigating, the problem came out to be that push_back function had a direct argument making it do unnecessary copy operations where instead std::move would make the operation more direct. After making the changes the execution time of parser() fell into a reasonable percentage as in Figure 1.4 and Figure 1.5.

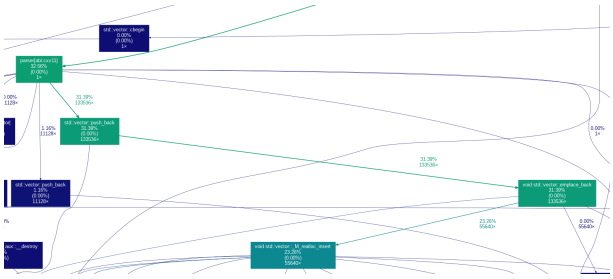


Figure 1.3 O3 optimised and compiled by g++

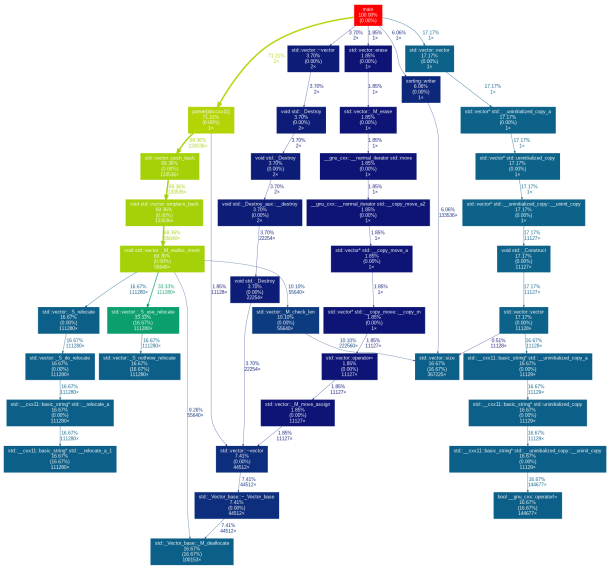


Figure 1.4 O3 optimised and compiled by g++

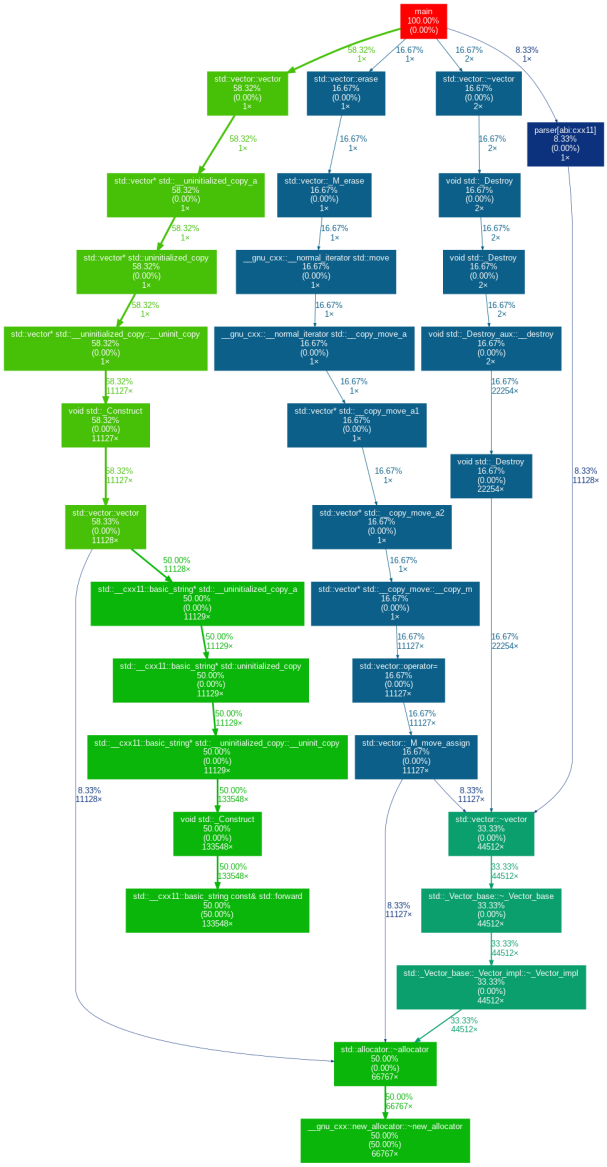


Figure 1.5 O3 optimised and compiled by clang++

The optimisations made by the compilers play a critical role at the performance of various parts of the code. Parser is optimised heavily in clang but not in gcc and they output different execution percentages. Apart from parser() the compiler also tries to optimize quicksort() function. In clang it is possible to view several optimizations made by the compiler on high level code with a tool named opt-viewer. In order to use it, -fsave-optimization-record -foptimization-record-file arguments should be given to clang. A yaml file containing optimisations will be generated. Opt-viewer can use the yaml file and generate html files. The opt-viewer is written in Python and it is in the llvm repository.

Figure 1.6 Optimisations of Clang provided by Opt-Viewer

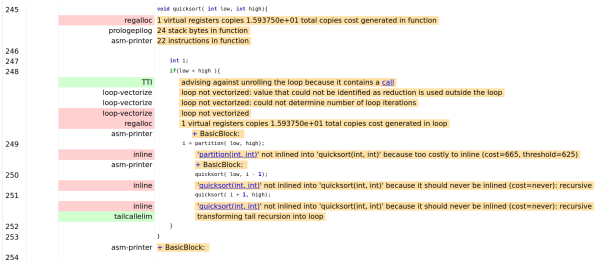


Figure 1.7 Optimisations of Clang provided by Opt-Viewer

The optimisations of Clang in the partition() function can be viewed at Figure 1.6 which are minimal. The optimisations on quicksort() can be seen at Figure 1.7 which includes tail call elimination.

The function call statistics are present in gprof but to have the statistics embedded in the code several counter variables are added and printed after the counting is finished. The time difference between entering the quicksort() is also printed. The time taken by quicksort is varying so the screenshots include 5 samples of the time difference. Similarly the screenshots for the second quicksort algorithm are given.

```
Using QS1 Algorithm with input size 11126
The reading of value is performed 1377448 times.
Swap is performed 51601 times.
Partition is performed 9815 times.
Sorted
writing to file
Duration of quicksort is 8067281 nanoseconds.

Using QS1 Algorithm with input size 11126
The reading of value is performed 1377448 times.
Swap is performed 51601 times.
Partition is performed 9815 times.
Sorted
writing to file
Duration of quicksort is 7342016 nanoseconds.

Using QS1 Algorithm with input size 11126
The reading of value is performed 1377448 times.
Swap is performed 51601 times.
Partition is performed 9815 times.
Sorted
writing to file
Duration of quicksort is 8430394 nanoseconds.

Using QS1 Algorithm with input size 11126
The reading of value is performed 1377448 times.
Swap is performed 51601 times.
Partition is performed 9815 times.
Sorted
writing to file
Duration of quicksort is 9075373 nanoseconds.

Using QS1 Algorithm with input size 11126
The reading of value is performed 1377448 times.
Swap is performed 51601 times.
Partition is performed 9815 times.
Sorted
writing to file
Duration of quicksort is 7830710 nanoseconds.
```

Figure 1.8 Duration time of quicksort execution and function counts with 100% input

```
Using QS1 Algorithm with input size 5562
The reading of value is performed 411793 times.
Swap is performed 24529 times.
Partition is performed 4847 times.
Sorted
writing to file
Duration of quicksort is 6403456 nanoseconds.

Using QS1 Algorithm with input size 5562
The reading of value is performed 411793 times.
Swap is performed 24529 times.
Partition is performed 4847 times.
Sorted
writing to file
Duration of quicksort is 3019504 nanoseconds.

Using QS1 Algorithm with input size 5562
The reading of value is performed 411793 times.
Swap is performed 24529 times.
Partition is performed 4847 times.
Sorted
writing to file
Duration of quicksort is 2902170 nanoseconds.

Using QS1 Algorithm with input size 5562
The reading of value is performed 411793 times.
Swap is performed 24529 times.
Partition is performed 4847 times.
Sorted
writing to file
Duration of quicksort is 3078485 nanoseconds.

Using QS1 Algorithm with input size 5562
The reading of value is performed 411793 times.
Swap is performed 24529 times.
Partition is performed 4847 times.
Sorted
writing to file
Duration of quicksort is 3437641 nanoseconds.
```

Figure 1.9 Duration time of quicksort execution and function counts with 50% input

```
Using QS1 Algorithm with input size 2779
The reading of value is performed 142923 times.
Swap is performed 13393 times.
Partition is performed 2386 times.
Sorted
writing to file
Duration of quicksort is 2543416 nanoseconds.

Using QS1 Algorithm with input size 2779
The reading of value is performed 142923 times.
Swap is performed 13393 times.
Partition is performed 2386 times.
Sorted
writing to file
Duration of quicksort is 2460857 nanoseconds.

Using QS1 Algorithm with input size 2779
The reading of value is performed 142923 times.
Swap is performed 13393 times.
Partition is performed 2386 times.
Sorted
writing to file
Duration of quicksort is 2425951 nanoseconds.

Using QS1 Algorithm with input size 2779
The reading of value is performed 142923 times.
Swap is performed 13393 times.
Partition is performed 2386 times.
Sorted
writing to file
Duration of quicksort is 2584868 nanoseconds.

Using QS1 Algorithm with input size 2779
The reading of value is performed 142923 times.
Swap is performed 13393 times.
Partition is performed 2386 times.
Sorted
writing to file
Duration of quicksort is 2311909 nanoseconds.
```

Figure 1.10 Duration time of quicksort execution and function counts with 25% input

```
Using QS1 Algorithm with input size 1388
The reading of value is performed 50946 times.
Swap is performed 6051 times.
Partition is performed 1158 times.
Sorted
writing to file
Duration of quicksort is 972541 nanoseconds.

Using QS1 Algorithm with input size 1388
The reading of value is performed 50946 times.
Swap is performed 6051 times.
Partition is performed 1158 times.
Sorted
writing to file
Duration of quicksort is 925991 nanoseconds.

Using QS1 Algorithm with input size 1388
The reading of value is performed 50946 times.
Swap is performed 6051 times.
Partition is performed 1158 times.
Sorted
writing to file
Duration of quicksort is 1018426 nanoseconds.

Using QS1 Algorithm with input size 1388
The reading of value is performed 50946 times.
Swap is performed 6051 times.
Partition is performed 1158 times.
Sorted
writing to file
Duration of quicksort is 894062 nanoseconds.

Using QS1 Algorithm with input size 1388
The reading of value is performed 50946 times.
Swap is performed 6051 times.
Partition is performed 1158 times.
Sorted
writing to file
Duration of quicksort is 975022 nanoseconds.
```

Figure 1.11 Duration time of quicksort execution and function counts with 12% input

```
Using QS2 Algorithm with input size 11126
The reading of value is performed 173619 times.
Swap is performed 51948 times.
Partition is performed 6522 times.
Sorted
writing to file
Duration of quicksort is 3466338 nanoseconds.

Using QS2 Algorithm with input size 11126
The reading of value is performed 173619 times.
Swap is performed 51948 times.
Partition is performed 6522 times.
Sorted
writing to file
Duration of quicksort is 2246123 nanoseconds.

Using QS2 Algorithm with input size 11126
The reading of value is performed 173619 times.
Swap is performed 51948 times.
Partition is performed 6522 times.
Sorted
writing to file
Duration of quicksort is 2298935 nanoseconds.

Using QS2 Algorithm with input size 11126
The reading of value is performed 173619 times.
Swap is performed 51948 times.
Partition is performed 6522 times.
Sorted
writing to file
Duration of quicksort is 2432427 nanoseconds.

Using QS2 Algorithm with input size 11126
The reading of value is performed 173619 times.
Swap is performed 51948 times.
Partition is performed 6522 times.
Sorted
writing to file
Duration of quicksort is 2264754 nanoseconds.
```

Figure 1.12 Duration time of the second quicksort execution and function counts with 100% input

```
Using QS2 Algorithm with input size 5560
The reading of value is performed 86793 times.
Swap is performed 23299 times.
Partition is performed 3270 times.
Sorted
writing to file
Duration of quicksort is 2447893 nanoseconds.

Using QS2 Algorithm with input size 5560
The reading of value is performed 86793 times.
Swap is performed 23299 times.
Partition is performed 3270 times.
Sorted
writing to file
Duration of quicksort is 2231260 nanoseconds.

Using QS2 Algorithm with input size 5560
The reading of value is performed 86793 times.
Swap is performed 23299 times.
Partition is performed 3270 times.
Sorted
writing to file
Duration of quicksort is 2340381 nanoseconds.

Using QS2 Algorithm with input size 5560
The reading of value is performed 86793 times.
Swap is performed 23299 times.
Partition is performed 3270 times.
Sorted
writing to file
Duration of quicksort is 2097592 nanoseconds.

Using QS2 Algorithm with input size 5560
The reading of value is performed 86793 times.
Swap is performed 23299 times.
Partition is performed 3270 times.
Sorted
writing to file
Duration of quicksort is 1970694 nanoseconds.
```

Figure 1.13 Duration time of the second quicksort execution and function counts with 50% input

```
Using QS2 Algorithm with input size 2777
The reading of value is performed 38588 times.
Swap is performed 10314 times.
Partition is performed 1630 times.
Sorted
writing to file
Duration of quicksort is 1124594 nanoseconds.

Using QS2 Algorithm with input size 2777
The reading of value is performed 38588 times.
Swap is performed 10314 times.
Partition is performed 1630 times.
Sorted
writing to file
Duration of quicksort is 1106248 nanoseconds.

Using QS2 Algorithm with input size 2777
The reading of value is performed 38588 times.
Swap is performed 10314 times.
Partition is performed 1630 times.
Sorted
writing to file
Duration of quicksort is 1048182 nanoseconds.

Using QS2 Algorithm with input size 2777
The reading of value is performed 38588 times.
Swap is performed 10314 times.
Partition is performed 1630 times.
Sorted
writing to file
Duration of quicksort is 1091724 nanoseconds.

Using QS2 Algorithm with input size 2777
The reading of value is performed 38588 times.
Swap is performed 10314 times.
Partition is performed 1630 times.
Sorted
writing to file
Duration of quicksort is 1083359 nanoseconds.
```

Figure 1.14 Duration time of the second quicksort execution and function counts with 25% input

Input Size	Average Running Time (us)	Swap Count	Partition Count
1388	957	6051	1158
2779	2465	13393	2386
5562	3768	24529	4847
11126	8148	51601	9815

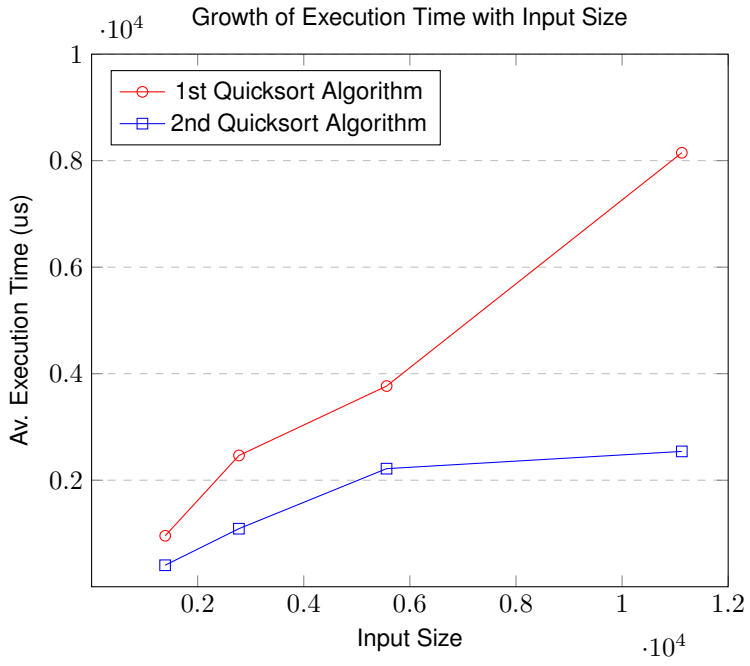
Table 1.1 Statistics regarding execution time and function call counts of the first algorithm

Input Size	Average Running Time (us)	Swap Count	Partition Count
1386	404	4594	840
2777	1090	10314	1630
5560	2217	23299	3270
11126	2541	51948	6522

Table 1.2 Statistics regarding execution time and function call counts of the second algorithm

```
Using QS2 Algorithm with input size 1386
The reading of value is performed 18399 times.
Swap is performed 4594 times.
Partition is performed 840 times.
Sorted
writing to file
Duration of quicksort is 214310 nanoseconds.

Using QS2 Algorithm with input size 1386
The reading of value is performed 18399 times.
Swap is performed 4594 times.
Partition is performed 840 times.
Sorted
writing to file
Duration of quicksort is 215526 nanoseconds.

Using QS2 Algorithm with input size 1386
The reading of value is performed 18399 times.
Swap is performed 4594 times.
Partition is performed 840 times.
Sorted
writing to file
Duration of quicksort is 524874 nanoseconds.

Using QS2 Algorithm with input size 1386
The reading of value is performed 18399 times.
Swap is performed 4594 times.
Partition is performed 840 times.
Sorted
writing to file
Duration of quicksort is 522128 nanoseconds.

Using QS2 Algorithm with input size 1386
The reading of value is performed 18399 times.
Swap is performed 4594 times.
Partition is performed 840 times.
Sorted
writing to file
Duration of quicksort is 548833 nanoseconds.
```

Figure 1.15 Duration time of the second quicksort execution and function counts with 12% input



As it can be seen at Table 1.1 and 1.2, smaller input sizes cause significant reduction in execution time.

1.0.2.1 Efficiency and Part2.d

I think the second algorithm is more efficient because every swap contributes to both sides of the pivot. However in the first algorithm swaps are done to move single elements rather than pairs.

The recurrence equation for quicksort is $T(n) = T(n_1)+T(n-1-n_1)+(n-1)$. As the partition is not guaranteed to split the array into two the, n_1 is given as the pivot index.

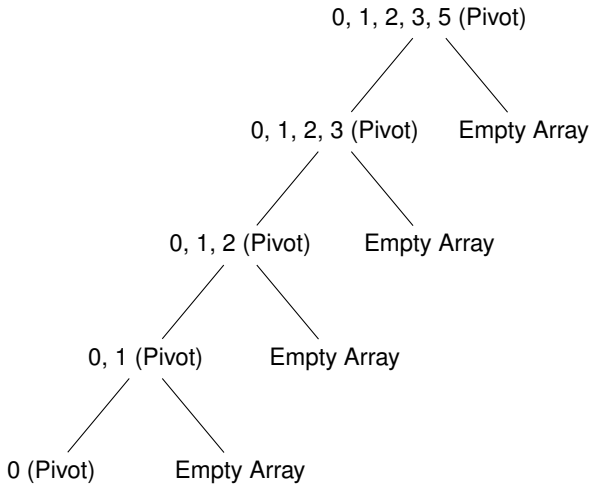
The worst case scenario is having one of the subarrays empty everytime which results the following recurrence equation$T(n) = T(0)+T(n-1)+(n-1) = T(n-1)+(n-1)$. $T(n-1)+(n-1) = T(n-2)+(n-2)+(n-1) = \sum_{i=1}^{n-1} i = (n-1)n/2$ which means the $O(n^2)$ is the average worst case asymptotic bound for quicksort.

The best case scenario occurs when there is no empty array and the divided arrays are of same size. For this condition, $n_1 = (n-1)/2$. The resulting recurrence equation is $T(n) = 2 * T(\frac{n-1}{2} + (n-1))$. It is the same as merge sort $O(n * \log n)$.

The average case scenario assumes uniform distribution of inputs. The pivot index n_1 can be 0 to n - 1 with equal probabaility. The slides of the course state that the $E[number of comparisons]$ is $O(n * \log n)$. The number of comparisons will dominate the runtime.

1.0.2.2 Worst Case Scenario and Part2.e

If the input is already sorted, with QS1 pivot is selected as the highest index element. So the partition results with the elements less than the pivot and and an empty array. Every partition results with an array smaller by one. As the n length array is traversed, partition runs n times. Partition includes n comparisons so the worst case runtime of $O(n^2)$ is achieved with this input.



The reason is the fixed pivot index, a random pivot could result more non-empty arrays. A checker at the beginning could help detect already sorted inputs. Also Knuth suggests using insertion sort after M sized files are produced by quicksort.

Chapter 2

Code Documentation

2.1 sorting Class Reference

Public Member Functions

- float getValue (int index)
- void swap (int index1, int index2)
- void extractor (vector< vector< string >> &toBeExtracted)
- bool checker ()
- void printer ()
- void writer (vector< string > header, vector< vector< string >> bookArray)

2.1.1 Detailed Description

to perform safer index based sorting a class with private arrays of the extracted numbers to be sorted and their referencing index array is created. Any function that needs to access these arrays except functions for the sorting are methods.

2.1.2 Member Function Documentation

2.1.2.1 checker()

```
bool sorting::checker ( )
```

Checker is used for testing the if the sorting was correct. It returns a boolean which is used for deciding to write to a file. It also useful for debugging.

2.1.2.2 extractor()

```
void sorting::extractor (
            vector< vector< string >> & toBeExtracted )
```

Extractor is used to minimise the usage of stof() function throughout the programme. It introduces a linear complexity which can be neglected. Unsorted_values array has the values extractec from the read vector of vector of strings. Also the sorted_keys are initialised as before the sorting every i index points to the i'th row.

2.1.2.3 getValue()

```
float sorting::getValue (
            int index )  [inline]
```

getValue() function returns the rating number at index refered by the sorted_keys array at index i. The index of value to be sorted is stored in another array which gets sorted.

2.1.2.4 printer()

```
void sorting::printer ( )
```

Printer() is used for debugging purposes, especially for smaller sized inputs.

2.1.2.5 swap()

```
void sorting::swap (
            int index1,
            int index2 )  [inline]
```

Swap is performed by swapping positions of value indices at sorted_keys array.

2.1.2.6 writer()

```
void sorting::writer (
            vector< string > header,
            vector< vector< string >> bookArray )
```

Writer() places the header of the csv first and then proceeds to read the vector of vector of strings. The read operation to the vector is performed non-sequentially. The row to be written is found by reading the sorted_keys array sequentially. The overhead introduced by finding the vector index from the sorted_keys array has a linear complexity as it is done once in the outer loop and the value is used throughout the inner loop.

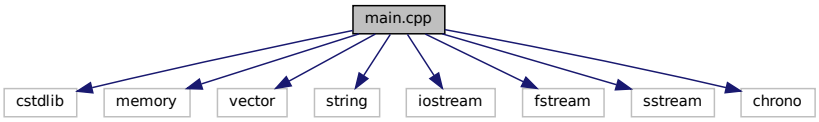
The documentation for this class was generated from the following file:

- main.cpp

2.2 main.cpp File Reference

```
#include <cstdlib>
#include <memory>
#include <vector>
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
```

Include dependency graph for main.cpp:



Classes

- class sorting

Macros

- #define stats 1

Functions

- vector< vector< string > > parser ()
- int tb_partition (int low, int high)
- void tb_quicksort (int low, int high)
- int partition (int low, int high)
- void quicksort (int low, int high)
- void partition_check (int low, int high, int pivot)
- int partition2 (int low, int high)
- void quicksort2 (int low, int high)
- int main ()

2.2.1 Function Documentation

2.2.1.1 main()

```
int main ( )
```

Parser returns the vector of vector of strings which is the parsed intermediate format of input csv file. The first row of header contains title information and has no data so it is stored in the header variable and the first row is erased in the vector. The low and high indices are calculated from the resulting data vector. Extractor initialises the sorting arrays. After sorting, the result is checked with checker. If it is correct the result is written to a csv file with writer(). For small input sizes the printer functions can be uncommented.

2.2.1.2 parser()

```
vector<vector<string> > parser ( )
```

Parser() reads the csv file in local directory named "books.csv" and feeds it into a vector of vector of strings. The vector of strings represent the rows. The outer vector is used as the file's internal representation in code.

Parser() opens the file and uses vector.push_back extensively. To prevent reallocation of memory extensively, vector.reserve() is used before editing the vector.

2.2.1.3 partition()

```
int partition (
            int low,
            int high )
```

The code for the Part I, partition function starts by walking i and j until a value bigger than the pivot is seen. The conditional j $<$ high - 1 is added for bound safety. After a value bigger than the pivot is seen j walks alone.Here is a conditional added: The swap to perform is made if only the new value is bigger than pivot. As j gets updated it can see values smaller than the pivot and it is wasteful to swap them.

The pseudocode allowed to update i in various ways. The method developed eventually was to make i walk until j or a value bigger than pivot was seen.

If j gets to the last element which is pivot, it gets swapped with i. As i is indicating the first element bigger than the pivot, it is a safe index to swap the pivot.

2.2.1.4 partition2()

```
int partition2 (
            int low,
            int high )
```

The code for Part II, the partition function starts by walking i from the low index and j from the high index. The pivot is selected randomly. In addition to the pseudocode, I decided to store the pivot in the low index at the beginning. i walks until seeing a value bigger than pivot and j walks backwards from the end until seeing a value smaller than pivot. If i did not exceed j, swap i and j.

2.2.1.5 partition_check()

```
void partition_check (
            int low,
            int high,
            int pivot )
```

2.2.1.6 quicksort()

```
void quicksort (
            int low,
            int high )
```

quicksort function of Part I. Since only the partition scheme changes the quicksort() functions of Part I and II are almost identical. The reason there is not an argument of array is that the value reading and swap operations are performed with a global object's methods. The objects can be made an argument as an improvement.

2.2.1.7 quicksort2()

```
void quicksort2 (
            int low,
            int high )
```

quicksort function of Part I. Since only the partition scheme changes the quicksort() functions of Part I and II are almost identical.

2.2.1.8 tb_partition()

```
int tb_partition (
            int low,
            int high )
```

The partition code is from the textbook (Introduction to Algorithms, Cormen) and used for debugging purposes.

2.2.1.9 tb_quicksort()

```
void tb_quicksort (
            int low,
            int high )
```

The quicksort code is from the textbook (Introduction to Algorithms, Cormen) and used for debugging purposes.